Workgroup: Network Working Group
Internet-Draft: draft-ietf-bpf-isa-01
Published: 4 March 2024
Intended Status: Standards Track
Expires: 5 September 2024
Authors: D. Thaler, Ed.

BPF Instruction Set Architecture (ISA)

Abstract

This document specifies the BPF instruction set architecture (ISA).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 5 September 2024.

Table of Contents

1. Documentation conventions

For brevity and consistency, this document refers to families of types using a shorthand syntax and refers to several expository, mnemonic functions when describing the semantics of instructions. The range of valid values for those types and the semantics of those functions are defined in the following subsections.

1.1. Types

This document refers to integer types with the notation *SN* to specify a type's signedness (*S*) and bit width (*N*), respectively.

S	Meaning
u	unsigned
s	signed

Table 1: Meaning of

signedness notation.

N	Bit width
8	8 bits
16	16 bits
32	32 bits
64	64 bits
128	128 bits

Table 2: Meaning of bit-width notation.

For example, *u32* is a type whose valid values are all the 32-bit unsigned numbers and *s16* is a types whose valid values are all the 16-bit signed numbers.

1.2. Functions

*htobe16: Takes an unsigned 16-bit number in host-endian format and returns the equivalent number as an unsigned 16-bit number in big-endian format.

*htobe32: Takes an unsigned 32-bit number in host-endian format and returns the equivalent number as an unsigned 32-bit number in big-endian format.

*htobe64: Takes an unsigned 64-bit number in host-endian format and returns the equivalent number as an unsigned 64-bit number in big-endian format.

*htole16: Takes an unsigned 16-bit number in host-endian format and returns the equivalent number as an unsigned 16-bit number in little-endian format.

*htole32: Takes an unsigned 32-bit number in host-endian format and returns the equivalent number as an unsigned 32-bit number in little-endian format.

*htole64: Takes an unsigned 64-bit number in host-endian format and returns the equivalent number as an unsigned 64-bit number in little-endian format.

*bswap16: Takes an unsigned 16-bit number in either big- or little-endian format and returns the equivalent number with the same bit width but opposite endianness.

- *bswap32: Takes an unsigned 32-bit number in either big- or little-endian format and returns the equivalent number with the same bit width but opposite endianness.

- *bswap64: Takes an unsigned 64-bit number in either big- or little-endian format and returns the equivalent number with the same bit width but opposite endianness.

1.3. Definitions

Sign Extend To *sign extend an* X *-bit number, A, to a* Y *-bit number, B ,* means to

1. Copy all X bits from *A* to the lower X bits of *B*.

2. Set the value of the remaining Y - X bits of *B* to the value of the most-significant bit of *A*.

Example

Sign extend an 8-bit number A to a 16-bit number B on a big-endian platform:

```
A:          10000110
B: 11111111 10000110
```

1.4. Conformance groups

An implementation does not need to support all instructions specified in this document (e.g., deprecated instructions). Instead, a number of conformance groups are specified. An implementation must support the base32 conformance group and may support additional conformance groups, where supporting a conformance group means it must support all instructions in that conformance group.

The use of named conformance groups enables interoperability between a runtime that executes instructions, and tools as such compilers that generate instructions for the runtime. Thus, capability discovery in terms of conformance groups might be done manually by users or automatically by tools.

Each conformance group has a short ASCII label (e.g., "base32") that corresponds to a set of instructions that are mandatory. That is, each instruction has one or more conformance groups of which it is a member.

This document defines the following conformance groups:

- *base32: includes all instructions defined in this specification unless otherwise noted.

- *base64: includes base32, plus instructions explicitly noted as being in the base64 conformance group.

- *atomic32: includes 32-bit atomic operation instructions (see Atomic operations (Section 4.3)).

- *atomic64: includes atomic32, plus 64-bit atomic operation instructions.

- *divmul32: includes 32-bit division, multiplication, and modulo instructions.

- *divmul64: includes divmul32, plus 64-bit division, multiplication, and modulo instructions.

- *packet: deprecated packet access instructions.

2. Instruction encoding

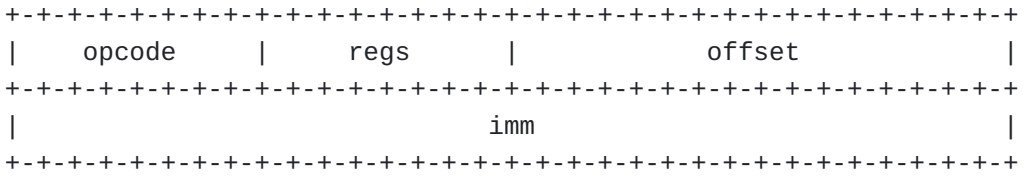
BPF has two instruction encodings:

- *the basic instruction encoding, which uses 64 bits to encode an instruction

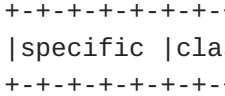
- *the wide instruction encoding, which appends a second 64 bits after the basic instruction for a total of 128 bits.

2.1. Basic instruction encoding

A basic instruction is encoded as follows:



opcode operation to perform, encoded as follows:



specific The format of these bits varies by instruction class

class The instruction class (see Instruction classes (Section 2.3))

regs The source and destination register numbers, encoded as follows on a little-endian host:

```
+-+-+-+-+-+-+-+-+
|src_reg|dst_reg|
+-+-+-+-+-+-+-+-+
```

and as follows on a big-endian host:

```
+-+-+-+-+-+-+-+-+
|dst_reg|src_reg|
+-+-+-+-+-+-+-+-+
```

src_reg the source register number (0-10), except where otherwise specified (64-bit immediate instructions (Section 4.4) reuse this field for other purposes)

dst_reg destination register number (0-10)

offset signed integer offset used with pointer arithmetic

imm signed integer immediate value

Note that the contents of multi-byte fields ('offset' and 'imm') are stored using big-endian byte ordering on big-endian hosts and little-endian byte ordering on little-endian hosts.

For example:

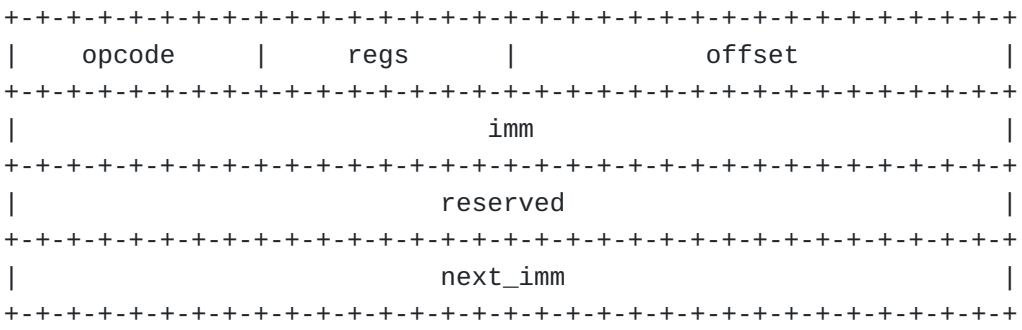
```
opcode                  offset imm          assembly
       src_reg dst_reg
07     0       1        00 00  44 33 22 11  r1 += 0x11223344 // little
       dst_reg src_reg
07     1       0        00 00  11 22 33 44  r1 += 0x11223344 // big
```

Note that most instructions do not use all of the fields. Unused fields shall be cleared to zero.

2.2. Wide instruction encoding

Some instructions are defined to use the wide instruction encoding, which uses two 32-bit immediate values. The 64 bits following the basic instruction format contain a pseudo instruction with 'opcode', 'dst_reg', 'src_reg', and 'offset' all set to zero.

This is depicted in the following figure:



opcode operation to perform, encoded as explained above

regs The source and destination register numbers, encoded as explained above

offset signed integer offset used with pointer arithmetic

imm signed integer immediate value

reserved unused, set to zero

next_imm second signed integer immediate value

2.3. Instruction classes

The three least significant bits of the 'opcode' field store the instruction class:

class	value	description	reference
LD	0x0	non-standard load operations	Load and store instructions (Section 4)
LDX	0x1	load into register operations	Load and store instructions (Section 4)
ST	0x2	store from immediate operations	Load and store instructions (Section 4)
STX	0x3	store from register operations	Load and store instructions (Section 4)
ALU	0x4	32-bit arithmetic operations	Arithmetic and jump instructions (Section 3)
JMP	0x5	64-bit jump operations	Arithmetic and jump instructions (Section 3)
JMP32	0x6	32-bit jump operations	Arithmetic and jump instructions (Section 3)
ALU64	0x7	64-bit arithmetic operations	Arithmetic and jump instructions (Section 3)

Table 3

3. Arithmetic and jump instructions

For arithmetic and jump instructions (ALU, ALU64, JMP and JMP32), the 8-bit 'opcode' field is divided into three parts:

```
+-+-+-+-+-+-+-+-+
|  code |s|class|
+-+-+-+-+-+-+-+-+
```

code the operation code, whose meaning varies by instruction class

s (source) the source operand location, which unless otherwise specified is one of:

source	value	description
K	0	use 32-bit 'imm' value as source operand
X	1	use 'src_reg' register value as source operand

Table 4

instruction class the instruction class (see Instruction classes (Section 2.3))

3.1. Arithmetic instructions

ALU uses 32-bit wide operands while ALU64 uses 64-bit wide operands for otherwise identical operations. ALU64 instructions belong to the base64 conformance group unless noted otherwise. The 'code' field encodes the operation as below, where 'src' and 'dst' refer to the values of the source and destination registers, respectively.

name	code	offset	description
ADD	0x0	0	dst += src
SUB	0x1	0	dst -= src
MUL	0x2	0	dst *= src
DIV	0x3	0	dst = (src != 0) ? (dst / src) : 0
SDIV	0x3	1	dst = (src != 0) ? (dst s/ src) : 0
OR	0x4	0	dst |= src
AND	0x5	0	dst &= src
LSH	0x6	0	dst <<= (src & mask)
RSH	0x7	0	dst >>= (src & mask)
NEG	0x8	0	dst = -dst
MOD	0x9	0	dst = (src != 0) ? (dst % src) : dst
SMOD	0x9	1	dst = (src != 0) ? (dst s% src) : dst
XOR	0xa	0	dst ^= src
MOV	0xb	0	dst = src
MOVSX	0xb	8/16/32	dst = (s8,s16,s32)src
ARSH	0xc	0	

name	code	offset	description
			sign extending (Section 1.3) dst >>= (src & mask)
END	0xd	0	byte swap operations (see Byte swap instructions (Section 3.2) below)

Table 5

Underflow and overflow are allowed during arithmetic operations, meaning the 64-bit or 32-bit value will wrap. If BPF program execution would result in division by zero, the destination register is instead set to zero. If execution would result in modulo by zero, for ALU64 the value of the destination register is unchanged whereas for ALU the upper 32 bits of the destination register are zeroed.

{ADD, X, ALU}, where 'code' = ADD, 'source' = X, and 'class' = ALU, means:

```
dst = (u32) ((u32) dst + (u32) src)
```

where '(u32)' indicates that the upper 32 bits are zeroed.

{ADD, X, ALU64} means:

```
dst = dst + src
```

{XOR, K, ALU} means:

```
dst = (u32) dst ^ (u32) imm
```

{XOR, K, ALU64} means:

```
dst = dst ^ imm
```

Note that most instructions have instruction offset of 0. Only three instructions (SDIV, SMOD, MOVSX) have a non-zero offset.

Division, multiplication, and modulo operations for ALU are part of the "divmul32" conformance group, and division, multiplication, and modulo operations for ALU64 are part of the "divmul64" conformance group. The division and modulo operations support both unsigned and signed flavors.

For unsigned operations (DIV and MOD), for ALU, 'imm' is interpreted as a 32-bit unsigned value. For ALU64, 'imm' is first sign extended (Section 1.3) from 32 to 64 bits, and then interpreted as a 64-bit unsigned value.

For signed operations (SDIV and SMOD), for ALU, 'imm' is interpreted as a 32-bit signed value. For ALU64, 'imm' is first sign extended

(Section 1.3) from 32 to 64 bits, and then interpreted as a 64-bit signed value.

Note that there are varying definitions of the signed modulo operation when the dividend or divisor are negative, where implementations often vary by language such that Python, Ruby, etc. differ from C, Go, Java, etc. This specification requires that signed modulo use truncated division (where -13 % 3 == -1) as implemented in C, Go, etc.:

```
a % n = a - n * trunc(a / n)
```

The MOVSX instruction does a move operation with sign extension. {MOVSX, X, ALU} sign extends (Section 1.3) 8-bit and 16-bit operands into 32 bit operands, and zeroes the remaining upper 32 bits. {MOVSX, X, ALU64} sign extends (Section 1.3) 8-bit, 16-bit, and 32-bit operands into 64 bit operands. Unlike other arithmetic instructions, MOVSX is only defined for register source operands (X).

The NEG instruction is only defined when the source bit is clear (K).

Shift operations use a mask of 0x3F (63) for 64-bit operations and 0x1F (31) for 32-bit operations.

3.2. Byte swap instructions

The byte swap instructions use instruction classes of ALU and ALU64 and a 4-bit 'code' field of END.

The byte swap instructions operate on the destination register only and do not use a separate source register or immediate value.

For ALU, the 1-bit source operand field in the opcode is used to select what byte order the operation converts from or to. For ALU64, the 1-bit source operand field in the opcode is reserved and must be set to 0.

class	source	value	description
ALU	TO_LE	0	convert between host byte order and little endian
ALU	TO_BE	1	convert between host byte order and big endian
ALU64	Reserved	0	do byte swap unconditionally

Table 6

The 'imm' field encodes the width of the swap operations. The following widths are supported: 16, 32 and 64. Width 64 operations

belong to the base64 conformance group and other swap operations belong to the base32 conformance group.

Examples:

{END, TO_LE, ALU} with imm = 16/32/64 means:

```
dst = htole16(dst)
dst = htole32(dst)
dst = htole64(dst)
```

{END, TO_BE, ALU} with imm = 16/32/64 means:

```
dst = htobe16(dst)
dst = htobe32(dst)
dst = htobe64(dst)
```

{END, TO_LE, ALU64} with imm = 16/32/64 means:

```
dst = bswap16(dst)
dst = bswap32(dst)
dst = bswap64(dst)
```

3.3. Jump instructions

JMP32 uses 32-bit wide operands and indicates the base32 conformance group, while JMP uses 64-bit wide operands for otherwise identical operations, and indicates the base64 conformance group unless otherwise specified. The 'code' field encodes the operation as below:

code	value	src_reg	description	notes
JA	0x0	0x0	PC += offset	{JA, K, JMP} only
JA	0x0	0x0	PC += imm	{JA, K, JMP32} only
JEQ	0x1	any	PC += offset if dst == src	
JGT	0x2	any	PC += offset if dst > src	unsigned
JGE	0x3	any	PC += offset if dst >= src	unsigned
JSET	0x4	any	PC += offset if dst & src	
JNE	0x5	any	PC += offset if dst != src	
JSGT	0x6	any	PC += offset if dst > src	signed
JSGE	0x7	any	PC += offset if dst >= src	signed
	0x8	0x0		

code	value	src_reg	description	notes
CALL			call helper function by address	{CALL, K, JMP} only, see Helper functions (Section 3.3.1)
CALL	0x8	0x1	call PC += imm	{CALL, K, JMP} only, see Program-local functions (Section 3.3.2)
CALL	0x8	0x2	call helper function by BTF ID	{CALL, K, JMP} only, see Helper functions (Section 3.3.1)
EXIT	0x9	0x0	return	{CALL, K, JMP} only
JLT	0xa	any	PC += offset if dst < src	unsigned
JLE	0xb	any	PC += offset if dst <= src	unsigned
JSLT	0xc	any	PC += offset if dst < src	signed
JSLE	0xd	any	PC += offset if dst <= src	signed

Table 7

The BPF program needs to store the return value into register R0 before doing an EXIT.

Example:

{JSGE, X, JMP32} means:

```
if (s32)dst s>= (s32)src goto +offset
```

where 's>=' indicates a signed '>=' comparison.

{JA, K, JMP32} means:

```
gotol +imm
```

where 'imm' means the branch offset comes from insn 'imm' field.

Note that there are two flavors of JA instructions. The JMP class permits a 16-bit jump offset specified by the 'offset' field, whereas the JMP32 class permits a 32-bit jump offset specified by the 'imm' field. A > 16-bit conditional jump may be converted to a < 16-bit conditional jump plus a 32-bit unconditional jump.

All CALL and JA instructions belong to the base32 conformance group.

3.3.1. Helper functions

Helper functions are a concept whereby BPF programs can call into a set of function calls exposed by the underlying platform.

Historically, each helper function was identified by an address encoded in the imm field. The available helper functions may differ for each program type, but address values are unique across all program types.

Platforms that support the BPF Type Format (BTF) support identifying a helper function by a BTF ID encoded in the imm field, where the BTF ID identifies the helper name and type.

3.3.2. Program-local functions

Program-local functions are functions exposed by the same BPF program as the caller, and are referenced by offset from the call instruction, similar to JA. The offset is encoded in the imm field of the call instruction. A EXIT within the program-local function will return to the caller.

4. Load and store instructions

For load and store instructions (LD, LDX, ST, and STX), the 8-bit 'opcode' field is divided as:

```
+-+-+-+-+-+-+-+-+
|mode |sz |class|
+-+-+-+-+-+-+-+-+
```

mode The mode modifier is one of:

mode modifier	value	description	reference
IMM	0	64-bit immediate instructions	64-bit immediate instructions (Section 4.4)
ABS	1	legacy BPF packet access (absolute)	Legacy BPF Packet access instructions (Section 4.5)
IND	2	legacy BPF packet access (indirect)	Legacy BPF Packet access instructions (Section 4.5)
MEM	3	regular load and store operations	Regular load and store operations (Section 4.1)
MEMSX	4	sign-extension load operations	Sign-extension load operations (Section 4.2)
ATOMIC	6	atomic operations	Atomic operations (Section 4.3)

Table 8

sz (size)

The size modifier is one of:

size	value	description
W	0	word (4 bytes)
H	1	half word (2 bytes)
B	2	byte
DW	3	double word (8 bytes)

Table 9

Instructions using DW belong to the base64 conformance group.

class The instruction class (see Instruction classes (Section 2.3))

4.1. Regular load and store operations

The MEM mode modifier is used to encode regular load and store instructions that transfer data between a register and memory.

{MEM, <size>, STX} means:

```
*(size *) (dst + offset) = src
```

{MEM, <size>, ST} means:

```
*(size *) (dst + offset) = imm
```

{MEM, <size>, LDX} means:

```
dst = *(unsigned size *) (src + offset)
```

Where '<size>' is one of: B, H, W, or DW, and 'unsigned size' is one of: u8, u16, u32, or u64.

4.2. Sign-extension load operations

The MEMSX mode modifier is used to encode sign-extension (Section 1.3) load instructions that transfer data between a register and memory.

{MEMSX, <size>, LDX} means:

```
dst = *(signed size *) (src + offset)
```

Where size is one of: B, H, or W, and 'signed size' is one of: s8, s16, or s32.

4.3. Atomic operations

Atomic operations are operations that operate on memory and can not be interrupted or corrupted by other access to the same memory region by other BPF programs or means outside of this specification.

All atomic operations supported by BPF are encoded as store operations that use the ATOMIC mode modifier as follows:

*{ATOMIC, W, STX} for 32-bit operations, which are part of the "atomic32" conformance group.

*{ATOMIC, DW, STX} for 64-bit operations, which are part of the "atomic64" conformance group.

*8-bit and 16-bit wide atomic operations are not supported.

The 'imm' field is used to encode the actual atomic operation. Simple atomic operation use a subset of the values defined to encode arithmetic operations in the 'imm' field to encode the atomic operation:

imm	value	description
ADD	0x00	atomic add
OR	0x40	atomic or
AND	0x50	atomic and
XOR	0xa0	atomic xor

Table 10

{ATOMIC, W, STX} with 'imm' = ADD means:

```
*(u32 *)(dst + offset) += src
```

{ATOMIC, DW, STX} with 'imm' = ADD means:

```
*(u64 *)(dst + offset) += src
```

In addition to the simple atomic operations, there also is a modifier and two complex atomic operations:

imm	value	description
FETCH	0x01	modifier: return old value
XCHG	0xe0 | FETCH	atomic exchange
CMPXCHG	0xf0 | FETCH	atomic compare and exchange

Table 11

The FETCH modifier is optional for simple atomic operations, and always set for the complex atomic operations. If the FETCH flag is

set, then the operation also overwrites src with the value that was in memory before it was modified.

The XCHG operation atomically exchanges src with the value addressed by dst + offset.

The CMPXCHG operation atomically compares the value addressed by dst + offset with R0. If they match, the value addressed by dst + offset is replaced with src. In either case, the value that was at dst + offset before the operation is zero-extended and loaded back to R0.

4.4. 64-bit immediate instructions

Instructions with the IMM 'mode' modifier use the wide instruction encoding defined in Instruction encoding (Section 2), and use the 'src_reg' field of the basic instruction to hold an opcode subtype.

The following table defines a set of {IMM, DW, LD} instructions with opcode subtypes in the 'src_reg' field, using new terms such as "map" defined further below:

src_reg	pseudocode	imm type	dst type
0x0	dst = (next_imm << 32) | imm	integer	integer
0x1	dst = map_by_fd(imm)	map fd	map
0x2	dst = map_val(map_by_fd(imm)) + next_imm	map fd	data pointer
0x3	dst = var_addr(imm)	variable id	data pointer
0x4	dst = code_addr(imm)	integer	code pointer
0x5	dst = map_by_idx(imm)	map index	map
0x6	dst = map_val(map_by_idx(imm)) + next_imm	map index	data pointer

Table 12

where

*map_by_fd(imm) means to convert a 32-bit file descriptor into an address of a map (see Maps (Section 4.4.1))

*map_by_idx(imm) means to convert a 32-bit index into an address of a map

*map_val(map) gets the address of the first value in a given map

*var_addr(imm) gets the address of a platform variable (see Platform Variables (Section 4.4.2)) with a given id

*code_addr(imm) gets the address of the instruction at a specified relative offset in number of (64-bit) instructions

*the 'imm type' can be used by disassemblers for display

*the 'dst type' can be used for verification and JIT compilation purposes

4.4.1. Maps

Maps are shared memory regions accessible by BPF programs on some platforms. A map can have various semantics as defined in a separate document, and may or may not have a single contiguous memory region, but the 'map_val(map)' is currently only defined for maps that do have a single contiguous memory region.

Each map can have a file descriptor (fd) if supported by the platform, where 'map_by_fd(imm)' means to get the map with the specified file descriptor. Each BPF program can also be defined to use a set of maps associated with the program at load time, and 'map_by_idx(imm)' means to get the map with the given index in the set associated with the BPF program containing the instruction.

4.4.2. Platform Variables

Platform variables are memory regions, identified by integer ids, exposed by the runtime and accessible by BPF programs on some platforms. The 'var_addr(imm)' operation means to get the address of the memory region identified by the given id.

4.5. Legacy BPF Packet access instructions

BPF previously introduced special instructions for access to packet data that were carried over from classic BPF. These instructions used an instruction class of LD, a size modifier of W, H, or B, and a mode modifier of ABS or IND. The 'dst_reg' and 'offset' fields were set to zero, and 'src_reg' was set to zero for ABS. However, these instructions are deprecated and should no longer be used. All legacy packet access instructions belong to the "packet" conformance group.

5. IANA Considerations

This document defines two sub-registries.

5.1. BPF Instruction Conformance Group Registry

This document defines a IANA sub-registry for BPF instruction conformance groups, as follows:

- *Name of the registry: BPF Instruction Conformance Groups
- *Name of the registry group: BPF Instructions
- *Required information for registrations: The values to appear in the entry fields.
- *Syntax of registry entries: Each entry has the following fields:
  - -name: alphanumeric label indicating the name of the conformance group
  - -description: brief description of the conformance group
  - -includes: any other conformance groups that are included from this group
  - -excludes: any other conformance groups that are excluded from this group.
  - -status: Permanent, Provisional, or Historical
  - -reference: a reference to the defining specification
- *Registration policy (see Section 4 of [RFC8126] for details):
  - -Permanent: Standards action or IESG Review
  - -Provisional: Specification required
  - -Historical: Specification required

Initial entries in this sub-registry are as follows:

name	description	includes	excludes	status	reference
atom32	32-bit atomic instructions	-	-	Permanent	RFCXXX Atomic operations (Section 4.3)
atom64	64-bit atomic instructions	atom32	-	Permanent	RFCXXX Atomic operations (Section 4.3)
base32		-	-	Permanent	RFCXXX

name	description	includes	excludes	status	reference
	32-bit base instructions				
base64	64-bit base instructions	base32	-	Permanent	RFCXXX
div32	32-bit division and modulo	-	-	Permanent	RFCXXX Arithmetic instructions (Section 3.1)
div64	64-bit division and modulo	div32	-	Permanent	RFCXXX Arithmetic instructions (Section 3.1)
packet	Legacy packet instructions	-	-	Historical	RFCXXX Legacy BPF Packet access instructions (Section 4.5)

Table 13

NOTE TO RFC-EDITOR: Upon publication, please replace RFCXXX above with reference to this document.

5.1.1. Adding instructions

A specification may add additional instructions to the BPF Instruction Set registry. Once a conformance group is registered with a set of instructions, no further instructions can be added to that conformance group. A specification should instead create a new conformance group that includes the original conformance group, plus any newly added instructions. Inclusion of the original conformance group is done via the "includes" column of the BPF Instruction Conformance Group Registry, and inclusion of newly added instructions is done via the "groups" column of the BPF Instruction Set Registry.

5.1.2. Deprecating instructions

Deprecating instructions that are part of an existing conformance group can be done by defining a new conformance group for the newly deprecated instructions, and the defining a new conformance group to supercede the existing conformance group containing the instructions, where the new conformance group includes the existing one and excludes the deprecated instruction group.

For example, if deprecating an instruction in an existing hypothetical group called "example", two new groups might be registered:

name	description	includes	excludes	status	reference
legacyexample	Legacy example instructions	-	-	Historical	(reference)
examplev2	Example instructions	example	legacyexample	Permanent	(reference)

Table 14

The BPF Instruction Set entries for the deprecated instructions would then be updated to add "legacyexample" to the set of groups for those instructions.

Finally, updated implementations that dropped support for the deprecated instructions would then be able to claim conformance to "examplev2" rather than "example".

5.2. BPF Instruction Set Registry

This document proposes a new IANA registry for BPF instructions, as follows:

- *Name of the registry: BPF Instruction Set
- *Name of the registry group: BPF Instructions
- *Required information for registrations: The values to appear in the entry fields.
- *Syntax of registry entries: Each entry has the following fields:
  - -opcode: a 1-byte value in hex format indicating the value of the opcode field
  - -src: either a value indicating the value of the src field, or "any"
  - -imm: either a value indicating the value of the imm field, or "any"
  - -offset: either a value indicating the value of the offset field, or "any"
  - -description: description of what the instruction does, typically in pseudocode

-groups: a list of one or more comma-separated conformance groups to which the instruction belongs

-reference: a reference to the defining specification

*Registration policy: New instructions require a new entry in the conformance group sub-registry and the same registration policies apply.

*Initial registrations: See the Appendix. Instructions other than those listed as deprecated are Permanent. Any listed as deprecated are Historical.

6. Acknowledgements

This draft was generated from instruction-set.rst in the Linux kernel repository, to which a number of other individuals have authored contributions over time, including Akhil Raj, Alexei Starovoitov, Brendan Jackman, Christoph Hellwig, Daniel Borkmann, Ilya Leoshkevich, Jiong Wang, Jose E. Marchesi, Kosuke Fujimoto, Shahab Vahedi, Tiezhu Yang, Will Hawkins, and Zheng Yejian, with review and suggestions by many others including Alan Jowett, Andrii Nakryiko, David Vernet, Jim Harris, Quentin Monnet, Song Liu, Shung-Hsi Yu, Stanislav Fomichev, and Yonghong Song.

7. Appendix

Initial values for the BPF Instruction sub-registry are given below. The descriptions in this table are informative. In case of any discrepancy, the reference is authoritative.

opcode	src_reg	offset	imm	description	groups	reference
0x00	0x0	0	any	(additional immediate value)	base64	64-bit immediate instructions (Section 4.4)
0x04	0x0	0	any	dst = (u32)((u32)dst + (u32)imm)	base32	Arithmetic instructions (Section 3.1)
0x05	0x0	any	0x00	goto +offset	base32	Jump instructions (Section 3.3)
0x06	0x0	0	any	goto +imm	base32	Jump instructions (Section 3.3)
0x07	0x0	0	any	dst += imm	base64	Arithmetic instructions (Section 3.1)
0x0c	any	0			base32	

opcode	src_reg	offset	imm	description	groups	reference
			0x00	dst = (u32)((u32)dst + (u32)src)		Arithmetic instructions (Section 3.1)
0x0f	any	0	0x00	dst += src	base64	Arithmetic instructions (Section 3.1)
0x14	0x0	0	any	dst = (u32)((u32)dst - (u32)imm)	base32	Arithmetic instructions (Section 3.1)
0x15	0x0	any	any	if dst == imm goto +offset	base64	Jump instructions (Section 3.3)
0x16	0x0	any	any	if (u32)dst == imm goto +offset	base32	Jump instructions (Section 3.3)
0x17	0x0	0	any	dst -= imm	base64	Arithmetic instructions (Section 3.1)
0x18	0x0	0	any	dst = (next_imm << 32) | imm	base64	64-bit immediate instructions (Section 4.4)
0x18	0x1	0	any	dst = map_by_fd(imm)	base64	64-bit immediate instructions (Section 4.4)
0x18	0x2	0	any	dst = map_val(map_by_fd(imm)) + next_imm	base64	64-bit immediate instructions (Section 4.4)
0x18	0x3	0	any	dst = var_addr(imm)	base64	64-bit immediate instructions (Section 4.4)
0x18	0x4	0	any	dst = code_addr(imm)	base64	64-bit immediate instructions (Section 4.4)
0x18	0x5	0	any	dst = map_by_idx(imm)	base64	64-bit immediate instructions (Section 4.4)
0x18	0x6	0	any	dst = map_val(map_by_idx(imm)) + next_imm	base64	64-bit immediate instructions (Section 4.4)
0x1c	any	0			base32	

opcode	src_reg	offset	imm	description	groups	reference
			0x00	dst = (u32)((u32)dst - (u32)src)		Arithmetic instructions (Section 3.1)
0x1d	any	any	0x00	if dst == src goto +offset	base64	Jump instructions (Section 3.3)
0x1e	any	any	0x00	if (u32)dst == (u32)src goto +offset	base32	Jump instructions (Section 3.3)
0x1f	any	0	0x00	dst -= src	base64	Arithmetic instructions (Section 3.1)
0x20	0x0	0	any	(deprecated, implementation-specific)	packet	Legacy BPF Packet access instructions (Section 4.5)
0x24	0x0	0	any	dst = (u32)(dst * imm)	divmul32	Arithmetic instructions (Section 3.1)
0x25	0x0	any	any	if dst > imm goto +offset	base64	Jump instructions (Section 3.3)
0x26	0x0	any	any	if (u32)dst > imm goto +offset	base32	Jump instructions (Section 3.3)
0x27	0x0	0	any	dst *= imm	divmul64	Arithmetic instructions (Section 3.1)
0x28	0x0	0	any	(deprecated, implementation-specific)	packet	Legacy BPF Packet access instructions (Section 4.5)
0x2c	any	0	0x00	dst = (u32)(dst * src)	divmul32	Arithmetic instructions (Section 3.1)
0x2d	any	any	0x00	if dst > src goto +offset	base64	Jump instructions (Section 3.3)
0x2e	any	any	0x00	if (u32)dst > (u32)src goto +offset	base32	Jump instructions (Section 3.3)
0x2f	any	0	0x00	dst *= src	divmul64	Arithmetic instructions (Section 3.1)
0x30	0x0	0	any	(deprecated, implementation-specific)	packet	Legacy BPF Packet access

opcode	src_reg	offset	imm	description	groups	reference
						instructions (Section 4.5)
0x34	0x0	0	any	dst = (u32)((imm != 0) ? ((u32)dst / (u32)imm) : 0)	divmul32	Arithmetic instructions (Section 3.1)
0x34	0x0	1	any	dst = (u32)((imm != 0) ? ((s32)dst s/ imm) : 0)	divmul32	Arithmetic instructions (Section 3.1)
0x35	0x0	any	any	if dst >= imm goto +offset	base64	Jump instructions (Section 3.3)
0x36	0x0	any	any	if (u32)dst >= imm goto +offset	base32	Jump instructions (Section 3.3)
0x37	0x0	0	any	dst = (imm != 0) ? (dst / (u32)imm) : 0	divmul64	Arithmetic instructions (Section 3.1)
0x37	0x0	1	any	dst = (imm != 0) ? (dst s/ imm) : 0	divmul64	Arithmetic instructions (Section 3.1)
0x3c	any	0	0x00	dst = (u32)((src != 0) ? ((u32)dst / (u32)src) : 0)	divmul32	Arithmetic instructions (Section 3.1)
0x3c	any	1	0x00	dst = (u32)((src != 0) ? ((s32)dst s/(s32)src) : 0)	divmul32	Arithmetic instructions (Section 3.1)
0x3d	any	any	0x00	if dst >= src goto +offset	base64	Jump instructions (Section 3.3)
0x3e	any	any	0x00	if (u32)dst >= (u32)src goto +offset	base32	Jump instructions (Section 3.3)
0x3f	any	0	0x00	dst = (src != 0) ? (dst / src) : 0	divmul64	Arithmetic instructions (Section 3.1)
0x3f	any	1	0x00	dst = (src != 0) ? (dst s/ src) : 0	divmul64	Arithmetic instructions (Section 3.1)
0x40	any	0	any	(deprecated, implementation-specific)	packet	Legacy BPF Packet access instructions (Section 4.5)
0x44	0x0	0	any	dst = (u32)(dst | imm)	base32	Arithmetic instructions (Section 3.1)
0x45	0x0	any	any	if dst & imm goto +offset	base64	

opcode	src_reg	offset	imm	description	groups	reference
						Jump instructions (Section 3.3)
0x46	0x0	any	any	if (u32)dst & imm goto +offset	base32	Jump instructions (Section 3.3)
0x47	0x0	0	any	dst |= imm	base64	Arithmetic instructions (Section 3.1)
0x48	any	0	any	(deprecated, implementation-specific)	packet	Legacy BPF Packet access instructions (Section 4.5)
0x4c	any	0	0x00	dst = (u32)(dst | src)	base32	Arithmetic instructions (Section 3.1)
0x4d	any	any	0x00	if dst & src goto +offset	base64	Jump instructions (Section 3.3)
0x4e	any	any	0x00	if (u32)dst & (u32)src goto +offset	base32	Jump instructions (Section 3.3)
0x4f	any	0	0x00	dst |= src	base64	Arithmetic instructions (Section 3.1)
0x50	any	0	any	(deprecated, implementation-specific)	packet	Legacy BPF Packet access instructions (Section 4.5)
0x54	0x0	0	any	dst = (u32)(dst & imm)	base32	Arithmetic instructions (Section 3.1)
0x55	0x0	any	any	if dst != imm goto +offset	base64	Jump instructions (Section 3.3)
0x56	0x0	any	any	if (u32)dst != imm goto +offset	base32	Jump instructions (Section 3.3)
0x57	0x0	0	any	dst &= imm	base64	Arithmetic instructions (Section 3.1)
0x5c	any	0	0x00	dst = (u32)(dst & src)	base32	Arithmetic instructions (Section 3.1)
0x5d	any	any	0x00	if dst != src goto +offset	base64	Jump instructions (Section 3.3)

opcode	src_reg	offset	imm	description	groups	reference
0x5e	any	any	0x00	if (u32)dst != (u32)src goto +offset	base32	Jump instructions (Section 3.3)
0x5f	any	0	0x00	dst &= src	base64	Arithmetic instructions (Section 3.1)
0x61	any	any	0x00	dst = *(u32 *)(src + offset)	base32	Load and store instructions (Section 4)
0x62	0x0	any	any	*(u32 *)(dst + offset) = imm	base32	Load and store instructions (Section 4)
0x63	any	any	0x00	*(u32 *)(dst + offset) = src	base32	Load and store instructions (Section 4)
0x64	0x0	0	any	dst = (u32)(dst << imm)	base32	Arithmetic instructions (Section 3.1)
0x65	0x0	any	any	if dst s> imm goto +offset	base64	Jump instructions (Section 3.3)
0x66	0x0	any	any	if (s32)dst s> (s32)imm goto +offset	base32	Jump instructions (Section 3.3)
0x67	0x0	0	any	dst <<= imm	base64	Arithmetic instructions (Section 3.1)
0x69	any	any	0x00	dst = *(u16 *)(src + offset)	base32	Load and store instructions (Section 4)
0x6a	0x0	any	any	*(u16 *)(dst + offset) = imm	base32	Load and store instructions (Section 4)
0x6b	any	any	0x00	*(u16 *)(dst + offset) = src	base32	Load and store instructions (Section 4)
0x6c	any	0	0x00	dst = (u32)(dst << src)	base32	Arithmetic instructions (Section 3.1)
0x6d	any	any	0x00	if dst s> src goto +offset	base64	Jump instructions (Section 3.3)
0x6e	any	any	0x00	if (s32)dst s> (s32)src goto +offset	base32	Jump instructions (Section 3.3)
0x6f	any	0	0x00	dst <<= src	base64	

opcode	src_reg	offset	imm	description	groups	reference
						Arithmetic instructions (Section 3.1)
0x71	any	any	0x00	dst = *(u8 *)(src + offset)	base32	Load and store instructions (Section 4)
0x72	0x0	any	any	*(u8 *)(dst + offset) = imm	base32	Load and store instructions (Section 4)
0x73	any	any	0x00	*(u8 *)(dst + offset) = src	base32	Load and store instructions (Section 4)
0x74	0x0	0	any	dst = (u32)(dst >> imm)	base32	Arithmetic instructions (Section 3.1)
0x75	0x0	any	any	if dst s>= imm goto +offset	base64	Jump instructions (Section 3.3)
0x76	0x0	any	any	if (s32)dst s>= (s32)imm goto +offset	base32	Jump instructions (Section 3.3)
0x77	0x0	0	any	dst >>= imm	base64	Arithmetic instructions (Section 3.1)
0x79	any	any	0x00	dst = *(u64 *)(src + offset)	base64	Load and store instructions (Section 4)
0x7a	0x0	any	any	*(u64 *)(dst + offset) = imm	base64	Load and store instructions (Section 4)
0x7b	any	any	0x00	*(u64 *)(dst + offset) = src	base64	Load and store instructions (Section 4)
0x7c	any	0	0x00	dst = (u32)(dst >> src)	base32	Arithmetic instructions (Section 3.1)
0x7d	any	any	0x00	if dst s>= src goto +offset	base64	Jump instructions (Section 3.3)
0x7e	any	any	0x00	if (s32)dst s>= (s32)src goto +offset	base32	Jump instructions (Section 3.3)
0x7f	any	0	0x00	dst >>= src	base64	Arithmetic instructions (Section 3.1)
0x84	0x0	0	0x00	dst = (u32)-dst	base32	

opcode	src_reg	offset	imm	description	groups	reference
						Arithmetic instructions (Section 3.1)
0x85	0x0	0	any	call helper function by address	base32	Helper functions (Section 3.3.1)
0x85	0x1	0	any	call PC += imm	base32	Program-local functions (Section 3.3.2)
0x85	0x2	0	any	call helper function by BTF ID	base32	Helper functions (Section 3.3.1)
0x87	0x0	0	0x00	dst = -dst	base32	Arithmetic instructions (Section 3.1)
0x94	0x0	0	any	dst = (u32)((imm != 0)? ((u32)dst % (u32)imm) : dst)	divmul32	Arithmetic instructions (Section 3.1)
0x94	0x0	1	any	dst = (u32)((imm != 0) ? ((s32)dst s% imm) : dst)	divmul32	Arithmetic instructions (Section 3.1)
0x95	0x0	0	0x00	return	base32	Jump instructions (Section 3.3)
0x97	0x0	0	any	dst = (imm != 0) ? (dst % (u32)imm) : dst	divmul64	Arithmetic instructions (Section 3.1)
0x97	0x0	1	any	dst = (imm != 0) ? (dst s% imm) : dst	divmul64	Arithmetic instructions (Section 3.1)
0x9c	any	0	0x00	dst = (u32)((src != 0)? ((u32)dst % (u32)src) : dst)	divmul32	Arithmetic instructions (Section 3.1)
0x9c	any	1	0x00	dst = (u32)((src != 0)? ((s32)dst s% (s32)src) :dst)	divmul32	Arithmetic instructions (Section 3.1)
0x9f	any	0	0x00	dst = (src != 0) ? (dst % src) : dst	divmul64	Arithmetic instructions (Section 3.1)
0x9f	any	1	0x00	dst = (src != 0) ? (dst s% src) : dst	divmul64	Arithmetic instructions (Section 3.1)
0xa4	0x0	0	any	dst = (u32)(dst ^ imm)	base32	

opcode	src_reg	offset	imm	description	groups	reference
						Arithmetic instructions (Section 3.1)
0xa5	0x0	any	any	if dst < imm goto +offset	base64	Jump instructions (Section 3.3)
0xa6	0x0	any	any	if (u32)dst < imm goto +offset	base32	Jump instructions (Section 3.3)
0xa7	0x0	0	any	dst ^= imm	base64	Arithmetic instructions (Section 3.1)
0xac	any	0	0x00	dst = (u32)(dst ^ src)	base32	Arithmetic instructions (Section 3.1)
0xad	any	any	0x00	if dst < src goto +offset	base64	Jump instructions (Section 3.3)
0xae	any	any	0x00	if (u32)dst < (u32)src goto +offset	base32	Jump instructions (Section 3.3)
0xaf	any	0	0x00	dst ^= src	base64	Arithmetic instructions (Section 3.1)
0xb4	0x0	0	any	dst = (u32) imm	base32	Arithmetic instructions (Section 3.1)
0xb5	0x0	any	any	if dst <= imm goto +offset	base64	Jump instructions (Section 3.3)
0xb6	0x0	any	any	if (u32)dst <= imm goto +offset	base32	Jump instructions (Section 3.3)
0xb7	0x0	0	any	dst = imm	base64	Arithmetic instructions (Section 3.1)
0xbc	any	0	0x00	dst = (u32) src	base32	Arithmetic instructions (Section 3.1)
0xbc	any	8	0x00	dst = (u32) (s32) (s8) src	base32	Arithmetic instructions (Section 3.1)
0xbc	any	16	0x00	dst = (u32) (s32) (s16) src	base32	Arithmetic instructions (Section 3.1)
0xbd	any	any	0x00	if dst <= src goto +offset	base64	

opcode	src_reg	offset	imm	description	groups	reference
						Jump instructions (Section 3.3)
0xbe	any	any	0x00	if (u32)dst <= (u32)src goto +offset	base32	Jump instructions (Section 3.3)
0xbf	any	0	0x00	dst = src	base64	Arithmetic instructions (Section 3.1)
0xbf	any	8	0x00	dst = (s64) (s8) src	base64	Arithmetic instructions (Section 3.1)
0xbf	any	16	0x00	dst = (s64) (s16) src	base64	Arithmetic instructions (Section 3.1)
0xbf	any	32	0x00	dst = (s64) (s32) src	base64	Arithmetic instructions (Section 3.1)
0xc3	any	any	0x00	lock *(u32 *)(dst + offset) += src	atomic32	Atomic operations (Section 4.3)
0xc3	any	any	0x01	src = atomic_fetch_add_32((u32 *)(dst + offset), src)	atomic32	Atomic operations (Section 4.3)
0xc3	any	any	0x40	lock *(u32 *)(dst + offset) |= src	atomic32	Atomic operations (Section 4.3)
0xc3	any	any	0x41	src = atomic_fetch_or_32((u32 *)(dst + offset), src)	atomic32	Atomic operations (Section 4.3)
0xc3	any	any	0x50	lock *(u32 *)(dst + offset) &= src	atomic32	Atomic operations (Section 4.3)
0xc3	any	any	0x51	src = atomic_fetch_and_32((u32 *)(dst + offset), src)	atomic32	Atomic operations (Section 4.3)
0xc3	any	any	0xa0	lock *(u32 *)(dst + offset) ^= src	atomic32	Atomic operations (Section 4.3)
0xc3	any	any	0xa1	src = atomic_fetch_xor_32((u32 *)(dst + offset), src)	atomic32	Atomic operations (Section 4.3)
0xc3	any	any	0xe1	src = xchg_32((u32 *)(dst + offset), src)	atomic32	Atomic operations (Section 4.3)
0xc3	any	any	0xf1		atomic32	

opcode	src_reg	offset	imm	description	groups	reference
				r0 = cmpxchg_32((u32 *) (dst + offset), r0, src)		Atomic operations (Section 4.3)
0xc4	0x0	0	any	dst = (u32)(dst s>> imm)	base32	Arithmetic instructions (Section 3.1)
0xc5	0x0	any	any	if dst s< imm goto +offset	base64	Jump instructions (Section 3.3)
0xc6	0x0	any	any	if (s32)dst s< (s32)imm goto +offset	base32	Jump instructions (Section 3.3)
0xc7	0x0	0	any	dst s>>= imm	base64	Arithmetic instructions (Section 3.1)
0xcc	any	0	0x00	dst = (u32)(dst s>> src)	base32	Arithmetic instructions (Section 3.1)
0xcd	any	any	0x00	if dst s< src goto +offset	base64	Jump instructions (Section 3.3)
0xce	any	any	0x00	if (s32)dst s< (s32)src goto +offset	base32	Jump instructions (Section 3.3)
0xcf	any	0	0x00	dst s>>= src	base64	Arithmetic instructions (Section 3.1)
0xd4	0x0	0	0x10	dst = htole16(dst)	base32	Byte swap instructions (Section 3.2)
0xd4	0x0	0	0x20	dst = htole32(dst)	base32	Byte swap instructions (Section 3.2)
0xd4	0x0	0	0x40	dst = htole64(dst)	base64	Byte swap instructions (Section 3.2)
0xd5	0x0	any	any	if dst s<= imm goto +offset	base64	Jump instructions (Section 3.3)
0xd6	0x0	any	any	if (s32)dst s<= (s32)imm goto +offset	base32	Jump instructions (Section 3.3)
0xd7	0x0	0	0x10	dst = bswap16(dst)	base32	Byte swap instructions (Section 3.2)
0xd7	0x0	0	0x20	dst = bswap32(dst)	base32	

opcode	src_reg	offset	imm	description	groups	reference
						Byte swap instructions (Section 3.2)
0xd7	0x0	0	0x40	dst = bswap64(dst)	base64	Byte swap instructions (Section 3.2)
0xdb	any	any	0x00	lock *(u64 *)(dst + offset) += src	atomic64	Atomic operations (Section 4.3)
0xdb	any	any	0x01	src = atomic_fetch_add_64((u64 *)(dst + offset), src)	atomic64	Atomic operations (Section 4.3)
0xdb	any	any	0x40	lock *(u64 *)(dst + offset) |= src	atomic64	Atomic operations (Section 4.3)
0xdb	any	any	0x41	src = atomic_fetch_or_64((u64 *)(dst + offset), src)	atomic64	Atomic operations (Section 4.3)
0xdb	any	any	0x50	lock *(u64 *)(dst + offset) &= src	atomic64	Atomic operations (Section 4.3)
0xdb	any	any	0x51	src = atomic_fetch_and_64((u64 *)(dst + offset), src)	atomic64	Atomic operations (Section 4.3)
0xdb	any	any	0xa0	lock *(u64 *)(dst + offset) ^= src	atomic64	Atomic operations (Section 4.3)
0xdb	any	any	0xa1	src = atomic_fetch_xor_64((u64 *)(dst + offset), src)	atomic64	Atomic operations (Section 4.3)
0xdb	any	any	0xe1	src = xchg_64((u64 *)(dst + offset), src)	atomic64	Atomic operations (Section 4.3)
0xdb	any	any	0xf1	r0 = cmpxchg_64((u64 *)(dst + offset), r0, src)	atomic64	Atomic operations (Section 4.3)
0xdc	0x0	0	0x10	dst = htobe16(dst)	base32	Byte swap instructions (Section 3.2)
0xdc	0x0	0	0x20	dst = htobe32(dst)	base32	Byte swap instructions (Section 3.2)
0xdc	0x0	0	0x40	dst = htobe64(dst)	base64	Byte swap instructions (Section 3.2)
0xdd	any	any	0x00	if dst s<= src goto +offset	base64	

opcode	src_reg	offset	imm	description	groups	reference
						Jump instructions (Section 3.3)
0xde	any	any	0x00	if (s32)dst s<= (s32)src goto +offset	base32	Jump instructions (Section 3.3)

Table 15

8. Normative References

[RFC8126] "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 8126, <https://www.rfc-editor.org/rfc/rfc8126>.

Author's Address

Dave Thaler (editor)
Redmond, WA 98052
United States of America

Email: dave.thaler.ietf@gmail.com